

Principled Programming

Introduction to Coding in Any Imperative Language

Tim Teitelbaum

Emeritus Professor

Department of Computer Science

Cornell University

Median

	0	1	2	3	4	n
A	10	20	30	40	50	

	0	1	2	3	4	5	n
A	10	20	30	40	50	60	

The *median* of an ordered array of n values is the middle value. If n is odd, this is $A[n/2]$; if n is even, we also opt for $A[n/2]$ rather than averaging the middle two values.

(Recall that $/$ is integer division when both operands are integers, where the fractional part is truncated. Thus, for example, $5/2$ is 2, and $6/2$ is 3.)

	0	1	2	3	4	n
A	50	30	10	40	20	

	0	1	2	3	4	5	n
A	50	30	10	60	20	40	

The *median* of an ordered array of n values is the middle value. If n is odd, this is $A[n/2]$; if n is even, we also opt for $A[n/2]$ rather than averaging the middle two values.

(Recall that $/$ is integer division when both operands are integers, where the fractional part is truncated. Thus, for example, $5/2$ is 2, and $6/2$ is 3.)

But what if the array is not ordered. How would you find the median then?

You could sort the array and select $A[n/2]$. But sorting requires $n \log n$ operations.

Is it possible to do better? Try it. You will find that everyday experience is no help.

	0	1	2	3	4	n
A	50	30	10	40	20	

	0	1	2	3	4	5	n
A	50	30	10	60	20	40	

The *median* of an ordered array of n values is the middle value. If n is odd, this is $A[n/2]$; if n is even, we also opt for $A[n/2]$ rather than averaging the middle two values.

(Recall that $/$ is integer division when both operands are integers, where the fractional part is truncated. Thus, for example, $5/2$ is 2, and $6/2$ is 3.)

But what if the array is not ordered. How would you find the median then?

You could sort the array and select $A[n/2]$. But sorting requires $n \log n$ operations.

Is it possible to do better? Try it. You will find that everyday experience is no help.

We need principles to follow in such cases.

	0	1	2	3	4	n
A	50	30	10	40	20	

	0	1	2	3	4	5	n
A	50	30	10	60	20	40	

Three principles that can help are:

-
- 👉 **Consider generalizing a problem when designing an algorithm.**
 - 👉 **Consider Divide and Conquer when designing an algorithm.**
 - 👉 **Consider recursion when designing an algorithm.**
-

We will use them to derive:

- An Average-Case Linear-Time Median Algorithm
- A Worst-Case Linear-Time Median Algorithm

It is **astounding** that it is possible to find the median of an unordered array of length n in linear time, i.e., time proportional to n .

The *median* of an ordered array of n values is the middle value. If n is odd, this is $A[n/2]$; if n is even, we opt for $A[n/2]$ rather than averaging the middle two values.

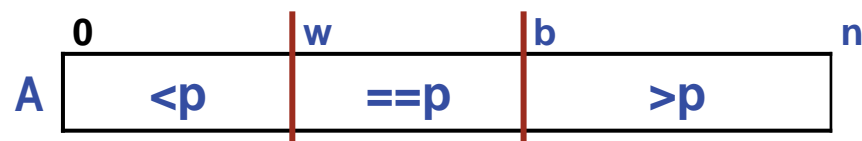
 **Consider generalizing a problem when designing an algorithm.**

Selection: Given a set of n rank-ordered values, select the j^{th} smallest value of the set.

Selection: Given a set of n rank-ordered values, select the j^{th} smallest value of the set.

☞ **Consider Divide and Conquer when designing an algorithm.**

Recall: Partitioning, based on the Dutch National Flag problem, for some pivot p :



$0 \leq j < w$. The j^{th} smallest value is the j^{th} smallest value of $A[0..w-1]$

$w \leq j < b$. The j^{th} smallest value is the pivot, p

$b \leq j < n$. The j^{th} smallest value is the $(j-b)^{\text{th}}$ smallest value in $A[b..n-1]$

Choose one of the three regions based on a Partition (Divide) and repeat (Conquer).

Start with the code for Partition, and morph it into QuickSelect:

```
/* Given A[L..R-1] and pivot value p, Partition(A,L,R,p) rearranges A[L..R-1]
   into all <p, then all ==p, then all >p. */
static void Partition( int A[], int L, int R, int p ) {
    <body of Partition>
} /* Partition */
```

 **Don't type if you can avoid it; clone. Cut and paste, then adapt.**

Start with the code for Partition, and morph it into QuickSelect:

```
/* Given A[L..n-1], and int  $0 \leq j < n$ , quick_select(A,n,j) returns the j-th  
   smallest value in A[0..n-1]. */  
static void QuickSelect( int A[], int L, int R, int p ) {  
    (body of Partition)  
} /* Partition */
```

 **Don't type if you can avoid it; clone. Cut and paste, then adapt.**

Change the name and header comment.

Start with the code for Partition, and morph it into QuickSelect:

```
/* Given A[L..n-1], and int  $0 \leq j < n$ , quick_select(A,n,j) returns the j-th
   smallest value in A[0..n-1]. */
static void QuickSelect( int A[], int n, int j ) {
    int L = 0; int R = n;
    int p = value-of-pivot;
    <body of Partition>
} /* QuickSelect */
```

 **Don't type if you can avoid it; clone. Cut and paste, then adapt.**

Move parameters L, R, and p into the body of QuickSelect, and introduce parameters n and j.

Start with the code for Partition, and morph it into QuickSelect:

```
/* Given A[L..n-1], and int  $0 \leq j < n$ , quick_select(A,n,j) returns the j-th
   smallest value in A[0..n-1]. */
static int QuickSelect( int A[], int n, int j ) {
    int L = 0; int R = n;
    int p = value-of-pivot;
    <body of Partition>
    return ____;
} /* QuickSelect */
```

 **Don't type if you can avoid it; clone. Cut and paste, then adapt.**

Change return type to **int**, and introduce a **return** statement for the result.

Could consider recursion, but it is not needed because we can just ...

```
/* Given A[L..n-1], and int  $0 \leq j < n$ , quick_select(A,n,j) returns the j-th
   smallest value in A[0..n-1]. */
static int QuickSelect( int A[], int n, int j ) {
    int L = 0; int R = n;
    int p = value-of-pivot;
    <body of Partition>
    return ____;
} /* QuickSelect */
```



INVARIANT

Update L , R , and p iteratively using the **INVARIANT** shown.

```

/* Given A[L..n-1], and int 0≤j<n, quick_select(A,n,j) returns the j-th
   smallest value in A[0..n-1]. */
static int QuickSelect( int A[], int n, int j ) {
  iterant { int L = 0; int R = n;
           int p = value-of-pivot;
           <body of Partition>
           return _____
         } /* QuickSelect */

```

```

/* Initialize. */
while ( not-finished ) {
  /* Compute. */           } iterant
  /* Go-on-to-next. */
}

```



Update L, R, and p iteratively using the **INVARIANT** shown.

/ Given A[L..n-1], and int $0 \leq j < n$, quick_select(A,n,j) returns the j-th smallest value in A[0..n-1]. */*

```
static int QuickSelect( int A[], int n, int j ) {
```

```
    int L = 0; int R = n;
```

```
    while ( not-finished ) {
```

```
        int p = value-of-pivot;
```

```
        <body of Partition>
```

```
        /* Go-on-to-next. */
```

```
    return ____;
```

```
 } /* QuickSelect */
```



Update L, R, and p iteratively using the **INVARIANT** shown.

```
/* Given A[L..n-1], and int 0≤j<n, quick_select(A,n,j) returns the j-th
   smallest value in A[0..n-1]. */
```

```
static int QuickSelect( int A[], int n, int j ) {
    int L = 0; int R = n;
    while ( not-finished ) {
        int p = value-of-pivot;
        <body of Partition>
        /* Go-on-to "<p" or ">p" region if j-th smallest there; else return p. */
    }
    return ____;
} /* QuickSelect */
```



Update L, R, and p iteratively using the **INVARIANT** shown.

/ Given A[L..n-1], and int $0 \leq j < n$, quick_select(A,n,j) returns the j-th smallest value in A[0..n-1]. */*

```
static int QuickSelect( int A[], int n, int j ) {
    int L = 0; int R = n;
    while ( not-finished ) {
        int p = value-of-pivot;
        <body of Partition>
        /* Go-on-to "<p" or ">p" region if j-th smallest there; else return p. */
        if ( j < w ) R = w;
        else if ( j < b ) return p;
        else L = b;
    }
    return ____;
} /* QuickSelect */
```




Update L, R, and p iteratively using the **INVARIANT** shown.

```
/* Given A[L..n-1], and int 0≤j<n, quick_select(A,n,j) returns the j-th
   smallest value in A[0..n-1]. */
```

```
static int QuickSelect( int A[], int n, int j ) {
    int L = 0; int R = n;
    while ( R-L > 1 ) {
        int p = value-of-pivot;
        <body of Partition>
        /* Go-on-to "<p" or ">p" region if j-th smallest there; else return p. */
        if ( j < w ) R = w;
        else if ( j < b ) return p;
        else L = b;
    }
    return A[j];
} /* QuickSelect */
```



Update L , R , and p iteratively using the **INVARIANT** shown.

```
/* Given A[L..n-1], and int 0≤j<n, quick_select(A,n,j) returns the j-th
   smallest value in A[0..n-1]. */
```

```
static int QuickSelect( int A[], int n, int j ) {
    int L = 0; int R = n;
    while ( R-L > 1 ) {
        int p = value-of-pivot;
        <body of Partition>
        /* Go-on-to "<p" or ">p" region if j-th smallest there; else return p. */
        if ( j < w ) R = w;
        else if ( j < b ) return p;
        else L = b;
    }
    return A[j];
} /* QuickSelect */
```

Q. Where was j ever updated?



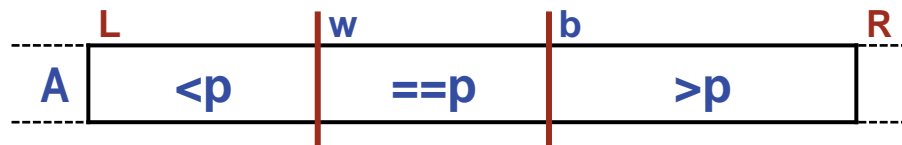
Update L, R, and p iteratively using the **INVARIANT** shown.

```
/* Given A[L..n-1], and int 0 ≤ j < n, quick_select(A,n,j) returns the j-th
   smallest value in A[0..n-1]. */
```

```
static int QuickSelect( int A[], int n, int j ) {
    int L = 0; int R = n;
    while ( R-L > 1 ) {
        int p = value-of-pivot;
        <body of Partition>
        /* Go-on-to "<p" or ">p" region if j-th smallest there; else return p. */
        if ( j < w ) R = w;
        else if ( j < b ) return p;
        else L = b;
    }
    return A[j];
} /* QuickSelect */
```

Q. Where was j ever updated?

A. Nowhere. Partitioning moved values so the j^{th} smallest ended up in A[j].



Update L , R , and p iteratively using the **INVARIANT** shown.

`/* Given $A[L..n-1]$, and $\text{int } 0 \leq j < n$, $\text{quick_select}(A, n, j)$ returns the j -th smallest value in $A[0..n-1]$. */`

```
static int QuickSelect( int A[], int n, int j ) {
    int L = 0; int R = n;
    while ( R-L > 1 ) {
        int p = (A[L]+A[R-1])/2;
        <body of Partition>
        /* Go-on-to "<p" or ">p" region if j-th smallest there; else return p. */
        if ( j < w ) R = w;
        else if ( j < b ) return p;
        else L = b;
    }
    return A[j];
} /* QuickSelect */
```

Performance: Pivots computed as $(A[L]+A[R-1])/2$

- Best case. On each iteration, pivot is (serendipitously) the **median** of $A[L..R-1]$, so region sizes reduced by $\frac{1}{2}$. Partitioning time is linear in size.
Total effort. $1 \cdot n + \frac{1}{2} \cdot n + \frac{1}{4} \cdot n + \dots = 2 \cdot n$, i.e., **linear** in n
- Worst case. On each iteration, pivot is (serendipitously) the **min or max** of $A[L..R-1]$, so region sizes reduced by **1**. Partitioning time is linear in size.
Total effort. $n + (n-1) + (n-2) + \dots + 1 = n \cdot (n-1)/2$, i.e., **quadratic** in n .
- Average case, i.e., summed over all permutations of values in $A[0..n-1]$.
Total effort. **Linear in n**
(offered without proof)

Bad News: `QuickSelect` can have quadratic-time performance on some arrays.

Imagine telling the widow:

But Mrs. Jones, on average the code would have been fast enough to have saved your husband's life.

Goal. Linear-time performance on every array.

Performance Goal: Pivots computed as _____ in the hope that

- Every case.

(1) On each iteration, region sizes reduced by constant ratio r .

Partitioning time is linear in region size.

Total effort for partitioning. $1 \cdot n + r \cdot n + r^2 \cdot n + r^3 \cdot n + \dots = n/(1-r)$

I.e., **linear** in n , *not counting time to compute the pivot.*

(2) On each iteration, the cost to compute the pivot is also linear in region size.

Thus, total effort, would be **linear** in n . In particular, even in the worst-case.

Performance Goal: Pivots computed as approximations to median of $A[L..R-1]$.

- Every case.

(1) On each iteration, region sizes reduced by constant ratio r .

Partitioning time is linear in region size.

Total effort for partitioning. $1 \cdot n + r \cdot n + r^2 \cdot n + r^3 \cdot n + \dots = n/(1-r)$

I.e., **linear** in n , *not counting time to compute the pivot.*

(2) On each iteration, the cost to compute the pivot is also linear in region size.

Thus, total effort, would be **linear** in n . In particular, even in the worst-case.

Idea: Pivots computed as approximations to median of $A[L..R-1]$.

Imagine that this array, with median 61:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
A	51	60	73	92	57	54	75	59	91	58	71	62	67	66	59	52	61	72	55	60	79

were laid out in a 3-high 2-D array in row major order:

51	60	73	92	57	54	75
59	91	58	71	62	67	66
59	52	61	72	55	60	79

The median of each column is shown in red.

Idea: Pivots computed as approximations to median of $A[L..R-1]$.

Imagine that this array, with median 61:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
A	51	60	73	92	57	54	75	59	91	58	71	62	67	66	59	52	61	72	55	60	79

were laid out in a 3-high 2-D array in row major order:

51	52	58	71	55	54	66
59	60	61	72	57	60	75
59	91	73	92	62	67	79

Now, imagine that each column were sorted, so its median comes to middle row.

The median of each column is shown in red.

Idea: Pivots computed as approximations to median of $A[L..R-1]$.

Imagine that this array, with median 61:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
A	51	60	73	92	57	54	75	59	91	58	71	62	67	66	59	52	61	72	55	60	79

were laid out in a 3-high 2-D array in row major order:

55	51	52	54	58	66	71
57	59	60	60	61	75	72
62	59	91	67	73	79	92

Next, imagine that the columns were sorted by their medians. The median of the medians is shown with a green background.

The median of each column is shown in red.

Idea: Pivots computed as approximations to median of $A[L..R-1]$.

Imagine that this array, with median 61:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
A	51	60	73	92	57	54	75	59	91	58	71	62	67	66	59	52	61	72	55	60	79

were laid out in a 3-high 2-D array in row major order:

55	51	52	54	58	66	71
57	59	60	60	61	75	72
62	59	91	67	73	79	92

Finally, color code the values:
 pink, if \leq median of medians
 blue, if \geq median of medians

The median of each column is shown in red.

Idea: Pivots computed as approximations to median of $A[L..R-1]$.

Imagine that this array, with median 61:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
A	51	60	73	92	57	54	75	59	91	58	71	62	67	66	59	52	61	72	55	60	79

were laid out in a 3-high 2-D array in row major order:

55	51	52	54	58	66	71
57	59	60	60	61	75	72
62	59	91	67	73	79	92

Finally, color code the values:
 pink, if \leq median of medians
 blue, if \geq median of medians

Choose the median of medians (60) as the pivot p , and partition A .

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
A	51	55	57	54	52	59	59	58	60	60	62	67	66	71	91	61	72	75	92	79	73

Idea: Pivots computed as approximations to median of $A[L..R-1]$.

Imagine that this array, with median 61:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
A	51	60	73	92	57	54	75	59	91	58	71	62	67	66	59	52	61	72	55	60	79

were laid out in a 3-high 2-D array in row major order:

55	51	52	54	58	66	71
57	59	60	60	61	75	72
62	59	91	67	73	79	92

We seek the median, i.e., the $n/2^{th}$ smallest ($n/2 = 21/2 = 10$), which falls into $>p$ region

Choose the median of medians (60) as the pivot p , and partition A .

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
A	51	55	57	54	52	59	59	58	60	60	62	67	66	71	91	61	72	75	92	79	73

Idea: Pivots computed as approximations to median of $A[L..R-1]$.

Imagine that this array, with median 61:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
A	51	60	73	92	57	54	75	59	91	58	71	62	67	66	59	52	61	72	55	60	79

were laid out in a 3-high 2-D array in row major order:

55	51	52	54	58	66	71
57	59	60	60	61	75	72
62	59	91	67	73	79	92

We seek the median, i.e., the $n/2^{th}$ smallest ($n/2 = 21/2 = 10$), which falls into $>p$ region, **eliminating** at least $2/3 \cdot 1/2 = 1/3$ the values.

Thus, the $>p$ region is no larger than $r = 1 - 1/3 = 2/3$ the size of the whole.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
A											62	67	66	71	91	61	72	75	92	79	73

Performance Goal: Pivots computed as approximations to median of $A[L..R-1]$.

- Every case.

(1) On each iteration, region sizes reduced by constant ratio r .

Partitioning time is linear in region size.

Total effort for partitioning. $n + r \cdot n + r^2 \cdot n + r^3 \cdot n + \dots = n/(1-r)$

I.e., **linear** in n , *not counting time to compute the pivot.*

(2) On each iteration, the cost to compute the pivot is also linear in region size.

Thus, total effort, would be **linear** in n . In particular, even in the worst-case.

Performance Goal: Pivots computed as **median of medians** of $A[L..R-1]$.

- Every case.

- (1) On each iteration, region sizes reduced by constant ratio r .

Partitioning time is linear in region size.

Total effort for partitioning. $n + (2/3) \cdot n + (2/3)^2 \cdot n + (2/3)^3 \cdot n + \dots = n / (1 - 2/3) = 3 \cdot n$

✓ I.e., **linear** in n , *not counting time to compute the pivot.*

- (2) On each iteration, the cost to compute the pivot is also linear in region size.

Thus, total effort, would be **linear** in n . In particular, even in the worst-case.

Performance Goal: Pivots computed as **median of medians of $A[L..R-1]$** .

- Every case.

- (1) On each iteration, region sizes reduced by constant ratio r .

Partitioning time is linear in region size.

Total effort for partitioning. $n + (2/3) \cdot n + (2/3)^2 \cdot n + (2/3)^3 \cdot n + \dots = n / (1 - 2/3) = 3 \cdot n$

✓ I.e., linear in n , *not counting time to compute the pivot.*

- (2) On each iteration, the cost to compute the pivot is also linear in region size.

But how will we compute the median of medians of $A[L..R-1]$?

Thus, total effort, would be linear in n . In particular, even in the worst-case.

Performance Goal: Pivots computed as median of medians of $A[L..R-1]$ using recursion, i.e., apply the worst-case median algorithm to the $n/3$ medians of groups of 3 elements.

 Consider recursion when designing an algorithm.

Performance Goal: Pivots computed as median of medians of $A[L..R-1]$ using recursion, i.e., apply the worst-case median algorithm to the $n/3$ medians of groups of 3 elements.

This works, but alas, there are too many groups of 3, so the total cost is super-linear.



Consider recursion when designing an algorithm.

Performance Goal: Pivots computed as median of medians of $A[L..R-1]$ using recursion, i.e., apply the worst-case median algorithm to the $n/3$ medians of groups of 3 elements.

This works, but alas, there are too many groups of 3, so the total cost is super-linear.

But don't lose heart. All is not lost, because ...

Performance Goal: Pivots computed as median of medians of $A[L..R-1]$ using recursion, i.e., apply the worst-case median algorithm to the $n/5$ medians of groups of 5 elements.

This works, and is linear.

Selection of a partition region eliminates at least $3/5 \cdot 1/2 = 3/10$ the values.

Thus, the selected region is no larger than $r = 1 - 3/10 = 7/10$ the size of the whole.

Total effort for partitioning. $n + (7/10) \cdot n + (7/10)^2 \cdot n + (7/10)^3 \cdot n + \dots = n / (1 - 7/10) = 3.33 \cdot n$

In effect, the reduction ratio r shrinks slightly (from $2/3$ to $3/10$), but the number of groups shrinks more than enough (from $n/3$ to $n/5$) to render the total linear.

Summary:

Presented three algorithm-design principles that can serve in lieu of everyday experience:

- ☞ Consider generalizing a problem when designing an algorithm.
- ☞ Consider Divide and Conquer when designing an algorithm.
- ☞ Consider recursion when designing an algorithm.

Used the principles to derive two algorithms for finding the median as a special case of finding the j^{th} smallest value in $A[0..n-1]$:

- Linear average-time performance
- Linear worst-case-time performance.